Rust Book - 7.0: Packages, Crates, and Modules

#rust Packages, Crates, and Modules

A key question when writing programs is*scope*: what names does the compiler know about at this location in the code? What functions am I allowed to call? What does this variable refer to?

Rust has a number of features related to scopes. This is sometimes called "the module system," but it encompases more than just modules:

- *Packages*are a Cargo feature that let you build, test, and share crates.
- *Crates*are a tree of modules that produce a library or executable.
- *Modules*and the*use*keyword let you control the scope and privacy of paths.
- A*path*is a way of naming an item such as a struct, function, or module.

Packages and Crates for Making Libraries and Executables

- A*crate*is a binary or library.
- The*crate root*is a source file that is used to know how to build a crate.
- A*package*has a*Cargo.toml*that describes how to build one or more crates. At most one crate in a package can be a library.

A package can contain zero or one library crates and as many binary crates as you'd like. There must be at least one crate (either a library or a binary) in a package.

- Modules, a way to organize code and control the privacy of paths
- Paths, a way to name items
- usea keyword to bring a path into scope
- pub, a keyword to make items public
- Renaming items when bringing them into scope with theaskeyword
- Using external packages
- Nested paths to clean up largeuselists
- Using the glob operator to bring everything in a module into scope
- How to split modules into individual files

Modules:

```
mod sound {
    mod instrument {
        mod woodwind {
            fn clarinet() {
                // Function body code goes here
            }
        }
    }
    mod voice {
    }
}
fn main() {
}
```

module tree:

```
crate
 └── sound
     └── instrument
        └── woodwind
     └── voice
```

Paths for Referring to an Item in the Module Tree

If we want to call a function, we need to know its*path*. "Path" is a synonym for "name" in a way, but it evokes that filesystem metaphor. Additionally, functions, structs, and other items may have multiple paths that refer to the same item, so "name" isn't quite the right concept.

A **path** can take two forms:

- An*absolute path*starts from a crate root by using a crate name or a literal `crate`.
- A*relative path*starts from the current module and uses `self`, `super`, or an identifier in the current module.

Both absolute and relative paths are followed by one or more identifiers separated by double colons (`::`).

Modules as the Privacy Boundary

Modules as the Privacy Boundary

…the syntax of modules can be used for organization. There’s another reason Rust has modules: modules are the **privacy boundary** in Rust. If you want to make an item like a function or struct private, you put it in a module.

- All items (functions, methods, structs, enums, modules, annd constants) are private by default.
- You can use the `pub` keyword to make an item public.
- You aren’t allowed to use private code defined in modules that are children of the current module.
- You are allowed to use any code defined in ancestor modules or the current module.

items without the `pub` keyword are private as you look “down” the module tree from the current module, but items without the `pub` keyword are public as you look “up” the tree from the current module

Starting Relative Paths withsuper

Starting Relative Paths withsuper

You can also construct relative paths beginning withsuper. Doing so is like starting a filesystem path with..: the path starts from the*parent*module, rather than the current module.

```
mod sound {
    mod instrument {
        fn clarinet() {
            super::breathe_in();
        }
    }

    fn breathe_in() {
        // Function body code goes here
    }
}
```

Usingpubwith Structs and Enums

Usingpubwith Structs and Enums

You can designate structs and enums to be public in a similar way as we've shown with modules and functions, with a few additional details.
If you usepubbefore a struct definition, you make the struct public. However, the struct's fields are still private. You can choose to make each field public or not on a case-by-case basis.

```
mod plant {
    pub struct Vegetable {
        pub name: String,
        id: i32,
    }

    impl Vegetable {
        pub fn new(name: &str) -> Vegetable {
            Vegetable {
                name: String::from(name),
                id: 1,
            }
        }
    }
}

fn main() {
    let mut v = plant::Vegetable::new("squash");

    v.name = String::from("butternut squash");
    println!("{} are delicious", v.name);

    // The next line won't compile if we uncomment it:
    // println!("The ID is {}", v.id);
}
```

if you make a public enum, all of its variants are public. You only need the `pub` before the `enum` keyword

The `use` Keyword to Bring Paths into a Scope

```rust
mod sound {
    pub mod instrument {
        pub fn clarinet() {
            // Function body code goes here
        }
    }
}

use crate::sound::instrument;

fn main() {
    instrument::clarinet();
    instrument::clarinet();
    instrument::clarinet();
}
```

Using `self` instead of `crate`

```rust
use self::sound::instrument;
```

Idiomatic `use` Paths for Functions vs. Other Items

```rust
use crate::sound::instrument::clarinet;

fn main() {
    clarinet();
    clarinet();
    clarinet();
}
```

Renaming Types Brought Into Scope with the `as` Keyword

```
use std::fmt::Result;
use std::io::Result as IoResult;

fn function1() -> Result {
}
fn function2() -> IoResult<()> {
}
```

Re-exporting Names with `pub use`

Re-exporting Names with `pub use`

When you bring a name into scope with the `use` keyword, the name being available in the new scope is private. If you want to enable code calling your code to be able to refer to the type as if it was defined in that scope just as your code does, you can combine `pub` and `use`. This technique is called*re-exporting*because you're bringing an item into scope but also making that item available for others to bring into their scope.

Nested Paths for Cleaning Up LargeuseLists

```
use std::cmp::Ordering;
use std::io;
// same as
use std::{cmp::Ordering, io};
```

```
use std::io;
use std::io::Write;
// same as
use std::io::{self, Write};
```

Separating Modules into Different Files

`src/main.rs`

```
mod sound;
```

```rust
fn main() {
    // Absolute path
    crate::sound::instrument::clarinet();
    // Relative path
    sound::instrument::clarinet();
}
```

`src/sound.rs`

```rust
pub mod instrument;
```

`src/instrument.rs`

```rust
pub fn clarinet() {
    // Function body code goes here
}
```

Rust provides ways to organize:

- your packages into crates,
- your crates into modules, and to refer to items defined in one module from another by specifying absolute or relative paths.

These paths can be brought into a scope with a `use` statement so that you can use a shorter path for multiple uses of the item in that scope.

Modules define code that's private by default, but you can choose to make definitions public by adding the `pub` keyword.